Big Data Network Flow Processing Using Apache Spark

Kamil Jeřábek and Ondřej Ryšavý
{ijerabek,rysavy}@fit.vutbr.cz
Brno University of Technology
Brno, Czech Republic

ABSTRACT

The increasing amount of traffic flows captured as a part of network monitoring activities makes the analysis more complicated. One of the goals for network traffic analysis is to identify malicious communication. In the paper, we present a new system for big data network flow classification and clustering. The proposed system is based on the popular big data engines such as Apache Spark and Apache Ignite. The conducted experiments demonstrate the feasibility of the proposed approach and show the possible scalability.

KEYWORDS

Big Data, Network flows, Apache Spark, Cassandra, Apache Ignite

ACM Reference Format:
Kamil Jeřábek and Ondřej Ryšavý. 2018. Big Data Network Flow Processing Using Apache Spark. In *Proceedings of Engineering Computer-based Systems (Submitted to ECBS'19).* ACM, New York, NY, USA, 9 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

Network monitoring is an essential part of a comprehensive security solution for existing data communications infrastructure. The devices collect various telemetry data that needs to be further processed. One of the frequently used representation is Netflow, which provides sufficiently detailed information about network traffic and is relatively economical in storage capacity. However, in some cases, it is required to store complete network communications in the form of packets captured. This is especially useful in situations where the identified threat needs to be thoroughly analyzed. The amount of packets captured is huge for today's networking technologies, bringing many complications. In addition to storage requirements, sufficient power is required to process stored data.

Operators capture and analyze network traffic for several reasons, including security threat identification, network troubleshooting, performance optimization, and so on. Capturing network traffic is usually more complex than obtaining other forms of digital data. Network data is only available on a network device for a limited time. An inappropriate collection method may result in data corruption or incomplete capture. Messages exchanged between applications are divided into segments. It is important to collect all relevant segments and combine them again into data flows. When collecting data on shared lines, there may be a huge amount of traffic from which only a fraction is relevant for analysis. In addition, many different protocols are used that require the use of appropriate decoding algorithms. Network packet analysis is therefore a computationally challenging task. Data capture is often implemented using special hardware. The processing of captured data is the task of various software tools.

Apache Spark [17] is an open source computing framework that offers a simple programming model suitable especially for batch processing of data flows. The key concept is to provide abstraction for data sharing represented by Resilient Distributed Datasets (RDD). Spark application performance is achieved by composing operations into a workflow that allows for parallel data processing and preserving data in memory whenever possible. Transformations process the data represented by RDD. All operations on the RDD are lazy. This allows Spark to analyze the entire graph representing the process pipeline and generate an effective execution plan. RDD can also be shared between calculations that provide significant acceleration. Apache Spark is complemented with several useful libraries, for instance, MLlib is machine learning library.

In this paper we deal with a system for processing large amounts of network communication data with the possibility of interactive querying. The system is focused on processing packets stored in pcap files, creating network flows and extracting relevant information from network IP flows. The main result of this work is a demonstration of the ability to process packet files using Apache Spark in combination with other technologies such as HDFS and Cassandra. Apache Spark seems to be a suitable platform for processing packet files. Each packet is a record that can be decoded independently of each other, and thus parallel processing can be used. Packets aggregate into streams based on a key calculated from several fields in the packet header. For each flow, additional information is usually extracted for further analysis, for example, the smallest, largest, or average packet size. The analysis then consists in searching for flows according to given criteria. Furthermore, it is possible to use machine learning to classify individual flows.

The paper is organized as follows. Section 2 provides an overview of related work. Section 3 introduces the system architecture and provides selected details about the computing cluster and processing pipeline settings. Section 4 describes the proposed experiments, the method of evaluation and the measured values. The last section discusses work results and identifies future research directions.

Submitted to ECBS'19, Sep 2-3, 2019, Bucharest, RO
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 000-0-0000-0000-0/00/00.
https://doi.org/00.0000/0000000.0000000

2 RELATED WORK

Network traffic analysis methods were implemented in a variety of network security tools. General purpose tools include network analyzers (Wireshark, TCP dump), IDS systems (Snort, Bro), fingerprinting tools (Nmap, p0f), and enterprise security analytics platforms (IBM Security Intelligence[1], RSA Security Analytics[2] or LogRhythm's Security Analytics [3]). Traditional tools are implemented for a conventional single machine computing architecture. Recent systems employ cloud or cluster-based technology to cope with a large amount of ingested data. Community projects Apache Metron [2] and Apache Spot [3] aim at the development of an industrial-strength solution for a complex network monitoring and security analytics. They are both implemented as distributed frameworks though using different technology. They can be used for security analysis of IT threats. They process a variety of data sources, e.g., firewall and application logs, emails, intrusion-detection reports, etc.

The need for big data analysis to security monitoring and threat detection was identified about a decade ago. Promises and challenges of applications of big data analysis to security domain were discussed since then [7]. Among other frameworks, the special attention was focused on the use of Map-Reduce approach and in particular, the Apache Spark platform. In the rest of this section, we overview the works closest to our contribution.

Lee and Lee [12] presented a Hadoop-based traffic monitoring system that performs IP, TCP, HTTP, and NetFlow analysis of multi-terabytes of traffic achieving a throughput of up to 14 Gbps. In their work [11], Kadam and Dhore also present a Hadoop-based traffic monitoring system that performs multiple types of analysis on a huge amount of Internet traffic in a scalable manner. Zhou et al. [19] identified basic tasks of exploratory analysis process of NetFlow dataset and describe the possible realization in the Hadoop framework. They also characterize the performance of the implemented process in two commonly used Hadoop deployments. To analyze the network traffic as big data, a scalable internet traffic analysis system was presented in [13]. It was shown that the system is capable of processing multi-terabytes packet dump files utilizing Apache Spark for data processing. ENTRADA is a distributed system that enables to store and interactively analyze network traffic data [16]. It was demonstrated for analysis of DNS traffic being able to process more than 100 TB of pcap files. System Hobbits, a Hadoop and Hive based traffic analysis tool for Internet Protocol (IP) and Transport Control Protocol (TCP) analysis of large-sized Internet traffic is presented in [10]. The system enables large-scale Internet traffic analysis on original libpcap files without preprocessing. The Hive-based interface simplifies writing analytical procedures. Recently, a new framework called hcap for analyzing PCAPs on Hadoop was presented [15]. Hcap aims to improve the performance of the experimental hadoop-pcap library developed by RIPE-NCC. In the study, the proposed framework was evaluated in terms of preprocessing, data retention, and query response time.

[1]http://www-03.ibm.com/security/solution/intelligence-big-data/
[2]http://www.emc.com/collateral/data-sheet/security-analytics-overview-ds.pdf.
[3]https://logrhythm.com/solutions/security/security-analytics/

3 SYSTEM ARCHITECTURE

Our proposed system for classification utilizes one of the most popular framework for big data processing – Apache Spark, which extends the Map-Reduce model to support more computations efficiently. Each Apache Spark application contains a driver process that and one or more executor processes. The driver process is primarily responsible for analyzing, distributing and scheduling work across the executors. The executor is a processing unit that is responsible for executing a task that the driver assigned to it and reporting the state back to the driver. Apache Spark differentiates two main node types, namely, the master node, and the worker node. The worker nodes run at least one executor process.

The Apache Spark system provides low-level APIs and structured APIs. The low-level APIs are represented by the data structure RDD (Resilient Distributed Dataset). It is a base for storing data objects. The structured APIs are represented by the Data Frame that is just merely a table of rows with a specified schema. The data structures are immutable. There are defined operations on the data structures called *transformations* and *actions*. The transformations are abstract operations among the structures that are subject to lazy evaluation. Apache Spark builds a plan and it does not do any transformation operation until action is triggered on the data structure. One of such actions is `count` operation.

Apache Spark breaks data into chunks called partitions. Partition is a collection of objects or rows depending on if we work with RDDs or Data Frames. Partitions are distributed across the cluster during the execution, and they are a subject to parallelism in Spark. The operations on each partition are run in parallel if there are enough cores. Nevertheless, if there is only one partition than Spark have a parallelism of only one. The Apache Spark core is written in Scala programming language, but we can write applications in other programming languages such as Java, Python or R. It also provides an opportunity to write SQL queries that are highly optimized [9].

The database for storing result data could be Apache Cassandra or Apache Ignite that provides the key-value based data storage. The Apache Cassandra database [1] is big data NoSQL distributed and decentralized storage. It is highly available and scales horizontally. It provides fault tolerance with the tunable consistency [8]. Apache Cassandra is often used as data storage for big data solutions in combination with Apache Spark.

Combining those technologies we can create a processing pipeline for the purpose of packet trace analysis. The design of the system consists of two parts: (i) packet capture processing component (see Figure 1) that decodes source packets and combines to network flows and (ii) network flow analysis component (see Figure 2) that performs firther analysis of identified conversations. divided into two parts.

The input data are stored in multiple `pcap` files on a Hadoop based distributed file system; in our case we use HDFS. The files are loaded into Apache Spark, and the first step is to parse raw frames into packets and group them to form network flows. In the next step, we compute and extract the statistical features for each flow. Result gives us a feature vector that can be stored in the database for future processing. Also, it is possible to perform immediate queries accessing data still available in the memory.

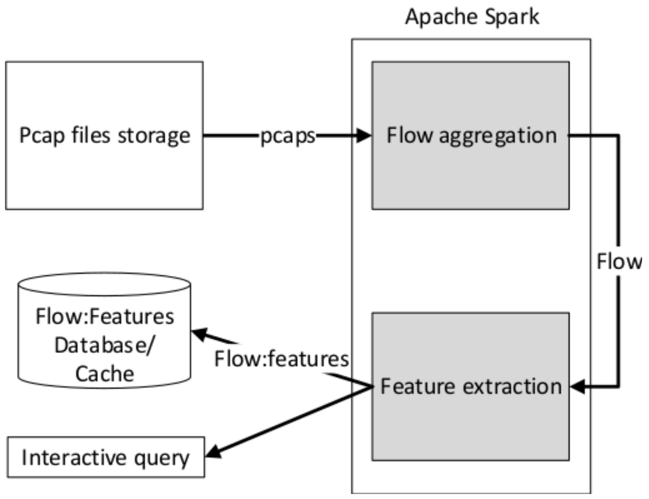


Figure 1: First (current) part of the system.

In this paper, we primarily focus on the first part demonstrating the capabilities of Apache Spark in network traffic processing. However, our goal is to implement the advanced traffic analysis system, which will be mainly realized by the second part. The interface between these two parts is represented by the distributed database that contains the flows with extracted statistical features. In the analysis, machine learning algorithms for flow classification and clustering will be employed. At first, we will train a classification model that will be able to distinguish malicious flows from normal flows. The malicious flows will be then clustered into different groups based on the application protocol or by the type of attack.

To extend the functionality of the system, we will use Apache Ignite [4]. The Apache Ignite provides multiple types of functionality. It is a scalable and fault-tolerant solution that can grow horizontally by adding new nodes. The Ignite can persist cache entries in RDBMS, and it also supports NoSQL like Cassandra or MongoDB.

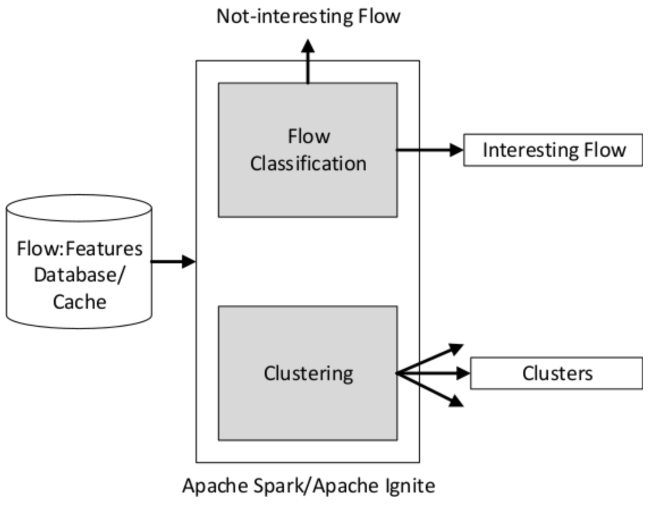


Figure 2: Second (future) part of the system.

The framework also provides Cache-as-a-Service for databases. One of the other features is that it can serve as an accelerator for Hadoop, and it also can share states in-memory across Spark jobs that is by default not possible in Spark. Moreover, Apache Ignite can do distributed computing and processing of never-ending streams of data. Machine learning library is included in this framework and provides a variety of configurable machine learning algorithms [18]. The Apache Ignite engine will serve as another framework in our system.

3.1 Cluster Setup

The design of the cluster is based on the Docker [5] container architecture. Docker containers were chosen according to the simplicity of deployment and environment isolation. The containers consume minimal staging and system sources. The docker containers also enable us to switch between different versions of the used frameworks. The platform utilized in experiments was Supermicro SuperTwin2 6026TT-TF server equipped with eight Intel (R) Xeon E5520 @ 2.26 GHz. The cluster consists of 4 nodes. Three nodes equipped with the 48 GB RAM and 16 CPU cores. One node equipped with the 23 GB RAM and 16 CPU cores. All nodes have installed 1 TB SSD disks that serve as a data storage for HDFS and Cassandra. We decided to use Docker Swarm for cluster management. The Docker Swarm is proprietary cluster manager for Docker. It is easy to set up and merely tunable solution.

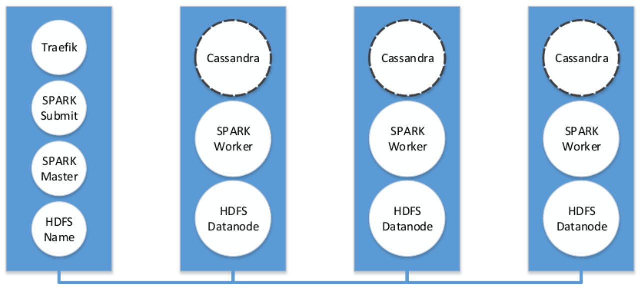


Figure 3: Docker cluster setup.

The spread of the images follows Figure 3. Each logical part of the different framework has its container. The Spark Worker container shares the same nodes with HDFS data node and a Cassandra node, to provide a closer connection between the processing unit and the data. This setup is set for three corresponding nodes. The last node is equipped with HDFS Namenode, Spark Master, Spark Submit container and Traefik container. The Spark Submit has its container from which the application is started. Spark application can be run in two modes, the client mode and cluster mode. In the client mode, the Spark driver resides in the container from which the application is started. The cluster mode runs the driver in one of the worker nodes. Choice of the modes should primarily depend on the distance of the spark submit from the cluster as the communication between the driver and the executors should be a bottleneck. We are running spark application in the client mode as the spark submit container resides on the same cluster. The Traefik [6] is a reverse proxy and load-balancer for the services running on the cluster. For experiments where we do the only query among extracted features, the Cassandra containers are not deployed.

3.2 Processing Pipeline

The task that we are dealing with this paper is to process network flows and get statistical features for further processing by the system. The data (captured network traffic) are stored in a distributed file system HDFS/IGFS. As the first step, we have to load files, extract raw frames (byte objects). The raw frames are then parsed and transformed into packets. The packets contain the parsed network and transport layer headers data together with other additional information. We use spark-ndx, the part of the processing platform presented in [14]. The spark-ndx module also provides a library functions for reading `pcap` files from the HDFS. For packet parsing, there are generated classes using Google Protocol Buffers. The only relevant features are extracted from the parsed packets into POJO class. Not relevant frames are filtered and deleted from the data set. The input data may contain frames that do not carry IP data. We are not interested in those.

The object containing packet information is aggregated into bi-directional flows. The statistical features are computed on the aggregated flows. The statistical features are then uploaded into the Cassandra database.

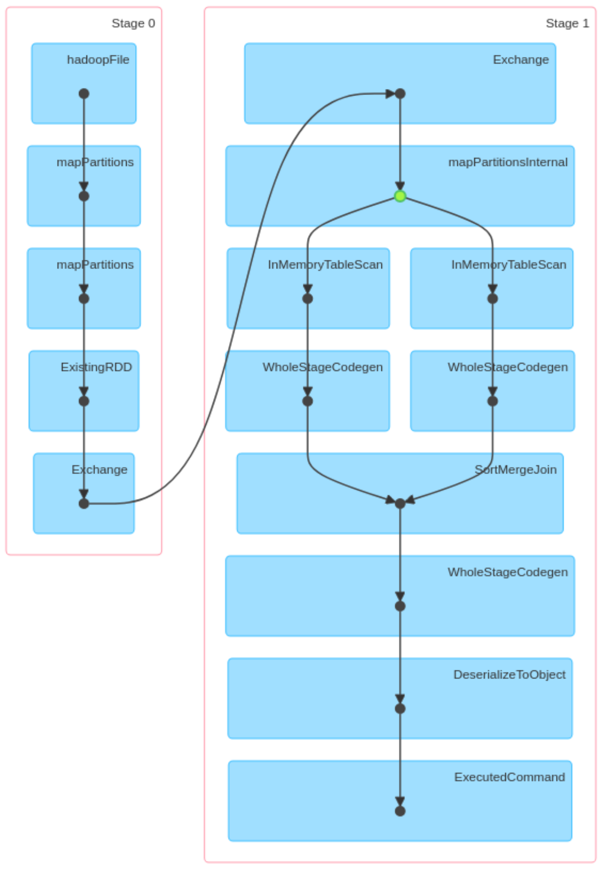


Figure 4: Directed Acyclic Graph generated by Spark.

The task was divided by Spark into two separated processing parts as it is depicted in Figure 4. The first part loads and works with data in a practical fashion way as with RDD. The second part work with data in the relational way hence the more suitable structure for processing is Data Frame. The second part is primarily transformed using Spark SQL.

4 EXPERIMENTS AND EVALUATION

This section describes experiments, measurement results and detailed analysis of the system performance on specified tasks. The tasks are of different level of complexity that begins from the most basic to more complex. While solving problems and running experiments we found some bottlenecks and places where the optimization should be useful. The Spark generates directed oriented graph with a sequence of operations divided into stages. The final DAG generated for our task is described in the previous section.

Main features that can utilize the Spark job and that we were utilizing are the usage of different programming languages, transformations on the data, cache, the number of partitions, a spread of workload between the different setup of executioners and cores.

The programming language we are using is Java. The different transformations such as `map`, `flatmap`, `mapPartitions` can lead to slower computation. The best results were obtained with `mapPartitions` that work with the data in a partition directly. During this step, the data are also filtered, so no more filtering afterward is needed.

Another feature is the cache. We are using SQL query among the packet attributes stored in Data Frame. There is a join in the SQL query that joins two different views on the same data in Data Frames. Even it is the same set of data on which the views are computed, the Spark generates two same stages as stage 0 for reading and parsing the `pcap` files from HDFS. Another key is the distribution of packet objects between the partitions. To prevent the data to be distributed on demand during the computation, we prevent that behavior by using a SQL query to distribute the packet rows by `flowKey`. The `flowKey` is an identifier that uniquely identifies all packets within the same bi-directional flow. The necessary data are redistributed. At this moment, the data can be cached. We used a native Spark cache that should keep the cached data in-memory. Alternatively, it should be cached on disk, HDFS or other third party cache solution. Among the cached data, we can do Spark queries effectively.

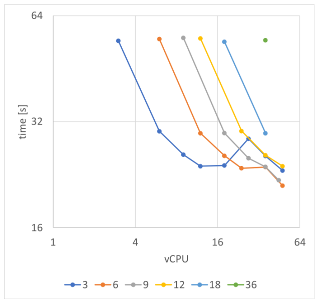
The partitions are directly related to the parallelism in Spark. Hence, their utilization can influence the computation efficiency of the job in different environments. The two parameters `spark.default.parallelism` together with `spark.sql.shuffle.partitions` can tune default parallelism in spark. The first one applies to RDD default partitioning. The second to inter Data Frame partitioning when using SQL, which is our case. The recommended values are 1-3 times the total number of cores, while the default Spark value is 200. For our query testing, we used the default value of 200.

The testing set of `pcap` files counts 231 files with 125000 frames each. The `pcap` files contain 27369774 packets collected from the honey-pot. The size of those files is 10GB. Many of the flows contain packets with no payload, which corresponds to some types of attacks. Hence the size variation of the files is significant. None of the files is bigger than 128 MB which is the default block size used in HDFS.

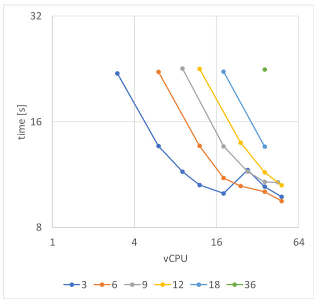
Time [s]	Description
25.392	Read from HDFS + parse packet + create pojo object + transform to DF.
19.112	Read from HDFS + parse packet + create pojo object.
14.921	Read from HDFS + parse packet.
3.242	Read from HDFS.

Table 1: Stage 0 analysis.

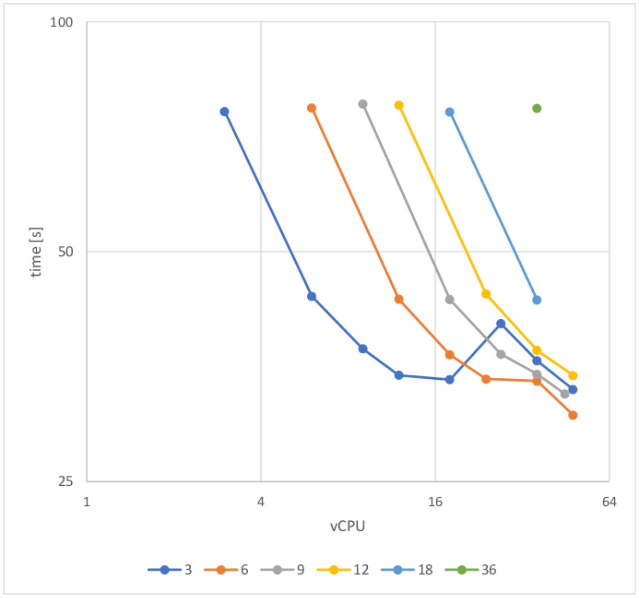
We also tested files of 100 MB size with a different number of frames. The results for stage 0 (covers reading files from HDFS and parsing packets) were worse than with the files divided by



(a) Stage 0



(b) Stage 1



(c) Both stages

Figure 5: Measurement of multiple feature extraction from flows of both stage 0 and stage 1 with different core utilization.

the specified number of frames. To investigate this behavior, we analyzed stage 0 in more detail.

The possible explanation of this behavior is that the bottleneck in stage 0 (file processing) is packet parsing according to Table 1. The results were measured using the `count` action on the different parts of the stage; the number of executors was 6 with 36 number of cores. The tested data were replicated to all data nodes. The more packets are in the file, the longer time the parsing consumes. The parsing should be simplified. The spark-ndx parser parses frames up to the application layer. Parsing everything up to the application layer would not be necessary for us in future processing. However, as we are in the experimentation phase, we do not know precisely what features will be useful. For now, we can extract more information from packets if needed. However, this will be the place for improvement.

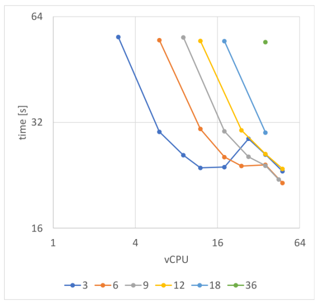
The significant impact on the computation time of the task has the setup of executors, cores and their numbers. The spread of the docker containers for each node is provided in Figure 3. We use those three fully equipped nodes for Spark workers, Cassandra and HDFS data nodes. The third node is dedicated for Spark master, HDFS name node, Traefik reverse proxy and Spark submit container. All jobs were submitted in client mode. The Spark submit container served as the driver for Spark.

Figure 5a shows the impact of the cores, its division between executioners on the processing time of described stage 0. The measurement starts with executioners that have only one core. The graph points to the fact that with more cores we get better results for reading and processing the data from HDFS. We can observe the different processing speed on the same amount of data for the different setup of executioners and cores assigned to them. As the stage 0 covers reading and parsing the frames from HDFS, we can see that with all cores involved in Spark job there is still performance improvement.

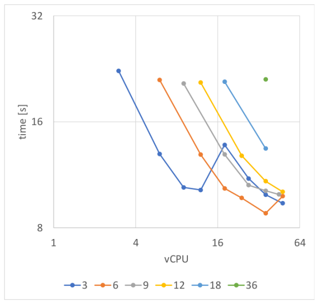
Name	Description
proto	Protocol
srcp	Source port
dstp	Destination port
packets	Number of packets in flow
size	Size of all packets in flow
paysize	Total size of payload in all packets in flow
duration	Flow duration
nopay	Number of packets without payload
avgps	Average packet size
minps	Minimal packet size
maxps	Maximal packet size
stddps	Standard deviation of packet size
avgpays	Average payload size
minpays	Minimal payload size
maxpays	Maximal payload size
stddpays	Standard deviation of payload size

Table 2: Flow feature names and descriptions.

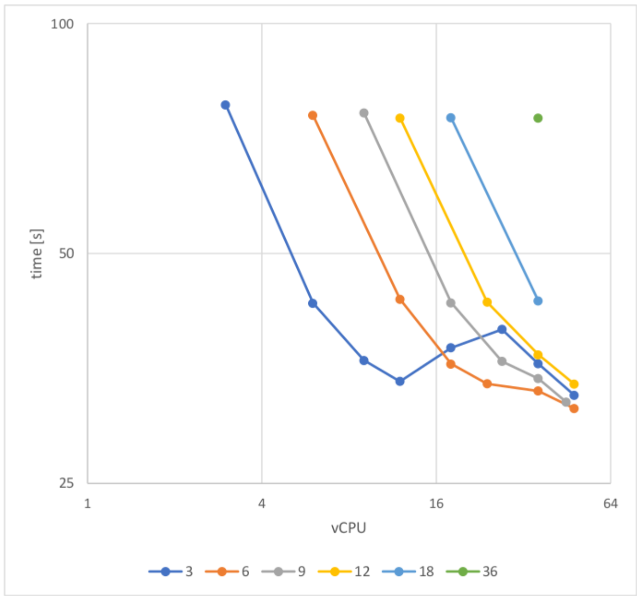
Figure 5b depicts the impact of the cores and its division between executioners on the processing time of the described stage 1. The results were measured after stage 0 completion. In this stage, we can see how well the application produces the output of the query. The features extracted from the flows are described in Table 2.



(a) Stage 0



(b) Stage 1



(c) Both stages

Figure 6: Processing performance of query among multiple feature extraction, both stage 0 and stage 1 with different core utilization.

The whole job measurement for this task, the stage 0 together with stage 1 are depicted in Figure 5c. We can see that the system scale is nearly linear, except for the anomaly using only three executioners. The fastest processing speed of the whole job working on 10GB data with setup of 6 executioners eight cores each was 30.55 seconds.

As another task, we consider one more Spark query above the previous one that selects only those flows that have content of zero length. Task measurement is depicted in Figure 6. The Spark optimizes those queries; hence there is no significant change in processing time. The fastest processing speed of the whole job working with 10GB data with setup of 6 executioners eight cores each was 31.33 seconds.

Fast information extraction of the flows data is the first part of our testing. This resulting data are provided to us once it is processed. One further task is also to save those processed data into persistent storage. It will enable us to perform queries later without processing again. For this task, we have chosen Cassandra distributed database. In this measurement, we perform a query for extracting the flow features as in the first measurement. In advance, when the data are processed, they are sent into Cassandra database. The Cassandra is run in a separated container on each worker node besides Spark Worker and HDFS Datanode.

```
CREATE TABLE flowfeaturess (
    srcaddr varchar,
    dstaddr varchar,
    srcp int,
    dstp int,
    proto varchar,
    size double,
    avgps float,
    minps int,
    maxps int,
    packets int,
    paysize double,
    avgpays float,
    minpays int,
    maxpays int,
    stddps float,
    stddpays float,
    duration int,
    nopayloadcnt int,
    PRIMARY KEY((srcaddr, dstaddr), srcp, dstp, proto)
);
```

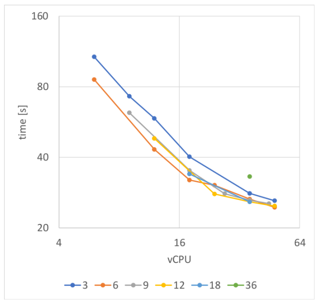
Figure 7: Cassandra table schema.

The design of the Cassandra table schema was tuned based on the flow features data extracted by the Spark job. The optimization regarding partitioning and clustering keys were done. The scheme is depicted in Figure 7.

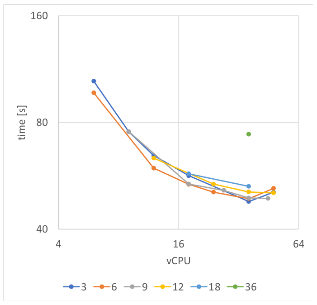
The scheme had a significant impact on the speed of writes into the database. The biggest bottleneck is the write to the database. Multiple parameters can tune the speed of the connector.

Another point is that the write using the connector is blocking operation. The Data-Stax connector for Cassandra is used. The results were produced with the default setting values for the connector. We observe that using the maximum number of cores can lead to slow down the computation time than with fewer cores. The slower computation time should be caused by involving the Cassandra database into receiving and storing the data. Hence in this setup should be better to use fewer cores.

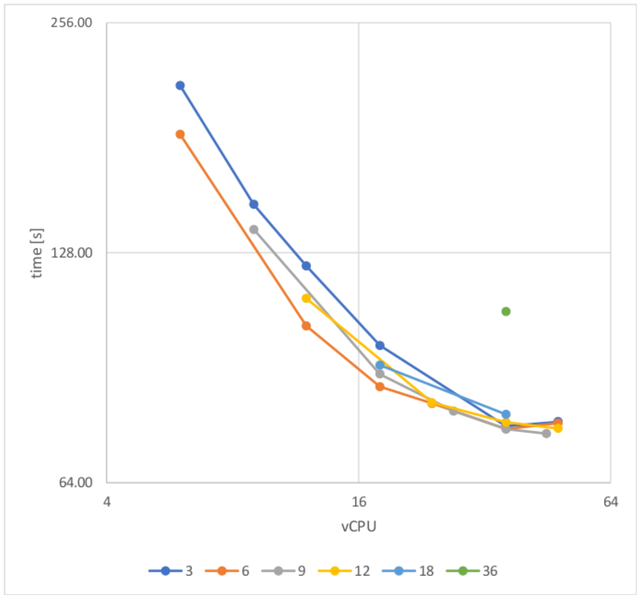
The final result of the measurements put the stage 0 and stage 1 together to a single job, and it is depicted in Figure 8. The graph correlates the previous deduction with the impact of the maximum number of cores for the task. The best speed for this task was 74.22 seconds for 45 cores and nine executors (5 cores per executor). However, in Figure 8a we see the change in case of usage 36 cores, where the performance is the best for most of the executioners. The



(a) Stage 0

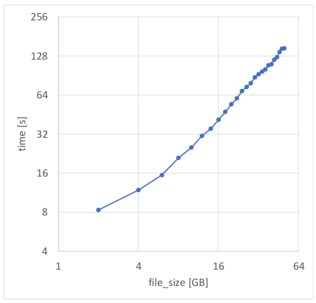


(b) Stage 1

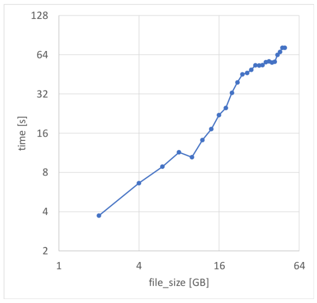


(c) Both stages

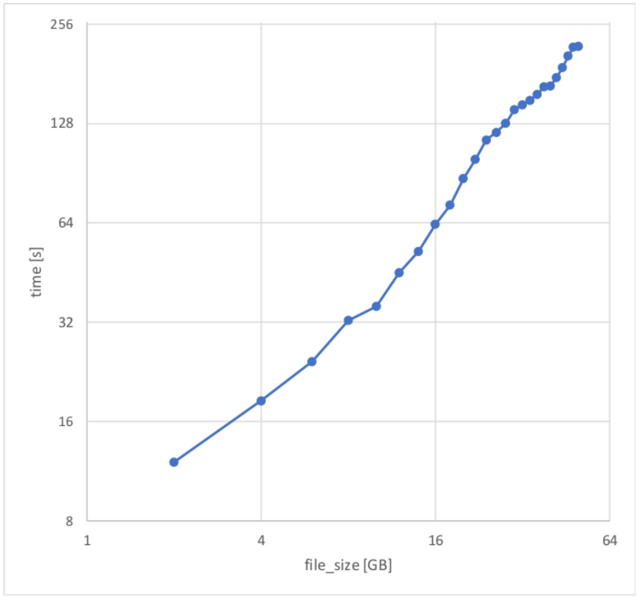
Figure 8: Processing performance of feature extraction with storing data into Cassandra, both stage 0 and stage 1 with different core utilization.



(a) Stage 0



(b) Stage 1



(c) Both stages

Figure 9: Processing performance of different file sizes, both stage 0 and stage 1 with different core utilization.

usage of more executioners decreases performance. It is caused by Cassandra database residing on the same nodes and cores.

All of the previous measurements provided were tested on 10GB file sizes. We want to test and provide a measurement that shows how the system scales with the processing of different data amount. We have chosen the same dataset for this test. The measurements start on 2GB of data and increases every time by 2GB up to 50GB. The dataset was divided into files that contained exactly 125, 000 frames each. The size of every file was less than 128MB which is the default block size used in HDFS. The same query as in the first measurement was used. As the setup for this test, we chosen 36 cores divided into six executioners (6 cores each). This setup was chosen because of the previous measurement with storing data into Cassandra database. We expect this setup for further usage together with this persistent storage.

The results are depicted in the set of Figures 9. We can consider that the system scales linearly with an increasing amount of data. Nevertheless, we are limited by the maximum amount of memory used on the nodes. The Spark carries everything in-memory, to be able to process the more massive amount of data the system has to be extended with more RAM.

The Table 3 shows concrete values from the measurements. We have chosen two setups to compare, the first one with 45 cores and nine executioners is the fastest in summary. The second one is the fastest in respect to stage 1 (data querying) and most stable regarding the deviation of the measurements. Each measurement was run five times, and the result is the average of the measured values.

From the results, we can observe that using the setup with more cores we can read and parse the data in higher speed, but the querying is slower than with fewer cores in a different setup. Even in more complicated tasks like more queries among the data we can see significant speed improvement with fewer cores setup than with more cores. The processing speed of both stages have closer

Stage	Setup	Time [s]	Throughput [Gbs]
Multiple features measurement			
0	**45 cores 9 exec**	**21.80**	**3.67**
0	36 cores 6 exec	23.76	3.37
1	45 cores 9 exec	10.75	7.44
1	**36 cores 6 exec**	**10.11**	**7.91**
both	**45 cores 9 exec**	**32.55**	**2.46**
both	36 cores 6 exec	33.87	2.36
Query among the multiple features measurement			
0	**45 cores 9 exec**	**22.00**	**3.64**
0	36 cores 6 exec	24.22	3.30
1	45 cores 9 exec	9.95	8.04
1	**36 cores 6 exec**	**8.80**	**9.09**
both	**45 cores 9 exec**	**31.94**	**2.50**
both	36 cores 6 exec	33.03	2.42
Cassandra storing measurement			
0	**45 cores 9 exec**	**25.39**	**3.15**
0	36 cores 6 exec	26.54	3.01
1	45 cores 9 exec	48.83	1.64
1	**36 cores 6 exec**	**48.73**	**1.64**
both	**45 cores 9 exec**	**74.22**	**1.08**
both	36 cores 6 exec	75.27	1.06

Table 3: Best results of the measurements with 10GB of data.

measurement time values with these more complex tasks, and results are more stable. Hence we decided to use 36 core setup with six executioners for further processing.

The processing throughput of the whole system for multiple feature extraction is 2.46 Gbs. The task with one query in advance is 2.50 Gbs; the speed improvement is caused by query optimization in Spark. The throughput of the last task with the Cassandra database reaches only 1.08 Gbs in the best case with default Cassandra connector setup.

5 CONCLUSION

A system for processing a large amount of captured network traffic was presented in this paper. The system is divided into two logical parts. The first part focuses on the feature extraction from the aggregated packets from network flows. The input to this part is pcap files, with the captured network traffic. The second part is intended to take the extracted flow features and classify them to decide whether the flow is malicious or normal. The Apache Cassandra is chosen as the persistent storage for flow features. This storage provides us an ability to use a different framework for classification (Spark or Ignite).

The presented design relies on existing big data technologies. We run the whole cluster on the docker container setup. This solution has valuable benefits such as environment isolation, the simplicity of deployment, and fast switching between different versions of frameworks. The presented experiments identified a bottleneck mainly in packet parsing algorithms that can be improved in the future. The task was optimized, and the cluster resources were utilized so that we were able to process multiple feature extraction on 10 GB of pcap files (27369774 packets) in 32.55 seconds using 45 cores on nine executors (5 cores per executor). In advance, the system was able to extract multiple features from the same amount of data and store them into the Cassandra database in 74, 219 seconds using the same amount of cores and executors. The system in this setup of 3 nodes where workers reside on and one dedicated node for spark driver etc. was able to process 2.46 Gbs for the extraction case and 1.08 Gbs for Cassandra case.

There were some problems during the configuration of Ignite on docker, so it was decided to optimize the solution in Spark together with Cassandra primarily. The speed of the HDFS is fine and the HDFS processed do not consume many resources when reading the data, in case of IGFS it can be worse because it consumes more RAM, which can interfere with data processing in Spark. On the other hand, it would be a better solution to use Apache Ignite instead of Cassandra as persistent storage. Apache Ignite is faster for writing the data which is the current bottleneck of the first stage. For future work, we plan to complete the implementation of the second part, which aims at malicious flow classification and clustering. Network traffic classification based on statistical features computed for network flows was demonstrated to be promising for malicious traffic identification and classification.

ACKNOWLEDGMENTS

This work was supported by Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project *IT4Innovations excellence in science* LQ1602; Ministry of Interior of the Czech Republic project *Integrated platform for analysis of digital data from security incidents* VI20172020062; and by BUT internal project *ICT tools, methods and technologies for smart cities* FIT-S-17-3964.

REFERENCES

[1] 2016. Apache Cassandra. https://cassandra.apache.org/
[2] 2016. Apache Metron: Real-Time Big Data Security. https://metron.incubator.apache.org/
[3] 2016. Apache Spot (incubating): A Community Approach to Fighting Cyber Threats. https://spot.incubator.apache.org/
[4] 2018. Apache Ignite. https://ignite.apache.org/
[5] 2018. Docker. https://www.docker.com/
[6] 2018. Traefik. https://traefik.io/
[7] Alvaro a. Cardenas, Pratyusa K. Manadhata, and Sreeranga P. Rajan. 2013. Big Data Analytics for Security. *IEEE Security & Privacy* (2013). https://doi.org/10.1109/MSP.2013.138
[8] Jeff Carpenter and Eben Hewitt. 2016. *Cassandra: The Definitive Guide: Distributed Data at Web Scale.* " O'Reilly Media, Inc.".
[9] Bill Chambers and Matei Zaharia. 2018. *Spark: the definitive guide: big data processing made simple.* " O'Reilly Media, Inc.".
[10] Abdeltawab M. Hendawi, Fatemah Alali, Xiaoyu Wang, Yunfei Guan, Tianshu Zhou, Xiao Liu, Nada Basit, and John A. Stankovic. 2016. Hobbits: Hadoop and Hive based Internet traffic analysis. In *2016 IEEE International Conference on Big Data, Big Data 2016.* IEEE, 2590–2599. https://doi.org/10.1109/BigData.2016.7840901
[11] Yogesh V Kadam and Prof Vaibhav Dhore. 2013. A Study on Scalable Internet Traffic Measurement and Analysis with Hadoop. *International Journal Of Engineering And Computer Science* (2013).
[12] Youngseok Yeonhee Lee and Youngseok Yeonhee Lee. 2013. Toward Scalable Internet Traffic Measurement and Analysis with Hadoop. *ACM SIGCOMM Computer Communication Review* 43, 1 (jan 2013), 5. https://doi.org/10.1145/2427036.2427038
[13] Alexey Lukashin, Leonid Laboshin, Vladimir Zaborovsky, and Vladimir Mulukha. 2014. *Distributed Packet Trace Processing Method for Information Security Analysis.* Springer International Publishing, Cham, 535–543. https://doi.org/10.1007/978-3-319-10353-2_49
[14] Marek Rychlý and Ondřej Ryšavý. 2018. Big Data Security Analysis with TARZAN Platform. *Journal of Cyber Security and Mobility* 8, 2 (2018), 165–188. https:

//doi.org/10.13052/jcsm2245-1439.822

[15] Miguel Zenon Nicanor Lerias Saavedra and William Emmanuel Yu. 2018. Towards large scale packet capture and network flow analysis on hadoop. *Proceedings - 2018 6th International Symposium on Computing and Networking Workshops, CANDARW 2018* (2018), 186–189. https://doi.org/10.1109/CANDARW.2018.00043

[16] M. Wullink, G. C. M. Moura, M. Muller, and C. Hesselman. 2016. ENTRADA: A high-performance network traffic data streaming warehouse. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 913–918. https://doi.org/10.1109/NOMS.2016.7502925

[17] Matei Zaharia, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, and Shivaram Venkataraman. 2016. Apache Spark. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

[18] Michael Zheludkov, Timur Isachenko, et al. 2017. *High Performance in-memory computing with Apache Ignite*. Lulu. com.

[19] X Zhou, M Petrovic, T Eskridge, M Carvalho, and X Tao. 2014. Exploring Netflow Data using Hadoop. In *2014 ASE BIGDATA/SOCIALCOM/CYBERSECURITY Conference*. 1–10. https://doi.org/q